RRANN: A Hardware Implementation of the Backpropagation Algorithm Using Reconfigurable FPGAs

James G. Eldredge and Brad L. Hutchings
Dept. of Electrical and Computer Eng.
Brigham Young University
Provo, UT 84602

1 Introduction

Neural network development has evolved to the point where they are being used in a wide range of useful applications. Some of these applications include loan underwriting [1], speech and text recognition [2, 3], and medical diagnoses [4]. For many of these applications, the execution speed provided by a software simulation running on a sequential, general purpose computer is sufficient. This is not true for all applications, and for this reason efforts have been made to speed-up neural network execution.

Special purpose hardware has been used to increase the execution speed for neural network algorithms. This includes massively parallel machines with thousands of processors [5, 6], and analog and digital ASICs (Application Specific Integrated Circuits) designed specifically for neural processing [7, 8]. Data representations range from high precision floating point numbers to long strings of electrical pulses [9]. Backpropagation is one of the most popular learning algorithms, and there have been several hardware projects devoted to it [10].

Field Programmable Gate Arrays (FPGAs) are an excellent technology for implementing neural networking hardware. Fast prototyping and reusability are general FPGA advantages that any application can exploit. In addition, neural networks can exploit the fact that RAM-based FPGAs may be reconfigured many times. This allows different topologies and learning algorithms to be implemented on the same FPGA. Also, many neural network algorithms (including backpropagation) lend themselves to run-time reconfiguration, where the FPGA is reconfigured during execution of the algorithm [11].

This paper presents the Run-Time Reconfiguration Artificial Neural Network (RRANN). RRANN is a hardware implementation of the backpropagation algorithm that is extremely scalable and makes efficient use of FPGA resources. One key feature is RRANN's ability to exploit parallelism in all stages of the backpropagation algorithm *including the stage where errors are propagated backward through the network.* This architecture has been designed and implemented on Xilinx XC3090 FPGAs, and its performance has been measured.

2 The Backpropagation Algorithm

The backpropagation learning algorithm provides a way to train a multilayered feed forward neural network. It begins by feeding input values forward through the network according to (1) and (2).

$$net_i = \sum_{j \in \mathcal{A}} O_j W_{ji} \quad \forall\, i : \; i \in \mathcal{B}, \tag{1}$$

where $\mathcal{A}$ is the set of all neurons in a layer, $\mathcal{B}$ is the set of all neurons in the next layer, O_j is the output (or activation) for neuron j, and W_{ji} is the weight assigned to the connection between neurons j and i. Equation (1) takes the output values (or activations) from a layer and feeds them forward to the next layer through weights. This operation is performed for each neuron in the next layer producing a *net* value for it. The *net* value is the weighted sum of all the activations of neurons in the previous layer, and each neuron's

net value is then applied to (2), known as the activation function, to produce that neuron's activation.

$$O_i = f(net_i) = \frac{1}{1+e^{-net_i}}. \tag{2}$$

After all neurons in the network have an activation value associated with them for a given input pattern, the backpropagation algorithm continues by finding errors for every non-input neuron. This process begins by finding the error for the neurons on the output layer according to (3).

$$\delta_i = f'(net_i)(T_i - O_i) \quad \forall\, i:\; i \in \mathcal{C}, \tag{3}$$

where $\mathcal{C}$ is the set of all output neurons, T_i is the target output for neuron i, $f'()$ is the first derivative of (2), and δ_i is the error for neuron i. The errors found for the output neurons are then propagated back to the previous layer so that errors can be assigned to neurons contained in hidden layers. This calculation is governed by (4)

$$\delta_i = f'(net_i) \sum_{j \in \mathcal{E}} \delta_j W_{ij} \quad \forall\, i:\; i \in \mathcal{D}, \tag{4}$$

where $\mathcal{D}$ is the set of all neurons on a non-input layer and $\mathcal{E}$ is the set of all neurons on the next layer. This calculation is repeated for every hidden layer in the network.

After every non-input neuron has an activation and error associated with it, the network's weights can be updated. First, the amount by which each weight should be changed is found by calculating (5).

$$\Delta W_{ij} = C O_i \delta_j \quad \forall\, i,j:\; i \in \mathcal{A},\; j \in \mathcal{B}, \tag{5}$$

where C, known as learning rate, is a constant controlling the size of weight changes, and ΔW_{ij} is the weight change between neuron i and j. The weight is changed by evaluating (6).

$$W_{ij_{t+1}} = W_{ij_t} + \Delta W_{ij}. \tag{6}$$

3 The RRANN Architecture

The RRANN architecture divides the backpropagation algorithm into the sequential execution of three stages known as feed forward, backpropagation, and update. The feed forward stage is responsible for taking input patterns and propagating them through the network assigning an activation (output) to every neuron according to Equations (1) and (2). The backpropagation stage (expressed by Equations (3) and (4)) finds the output errors and propagates them backward through the network in order to find errors for neurons contained in hidden layers. After every non-input neuron has been assigned an error value, the update stage (Equations (5) and (6)) begins operation. The update stage uses activation and error values found by the previous two stages to compute the amount by which weights should be changed and updates all weights with these changes. The completion of the update stage marks the end of the evaluation of one training pattern. This process is then repeated for all training patterns until the network is sufficiently trained.

RRANN contains a global controller occupying one FPGA and many neural processors occupying the balance of the available FPGAs (Figure 1). The global controller is mainly responsible for sequencing the execution of local hardware subroutines on the neural processors and for supplying key data to the neural processors (such as input patterns). Each neural processor contains six hardware neurons and local hardware subroutines implemented with state machines. The neural processors are responsible for performing all computations needed for the backpropagation algorithm. Each neural processor also has a local RAM associated with it. This RAM is used to store weight and target output data and as a scratch pad to buffer *net* and error values.

In order to use hardware efficiently, RRANN utilizes bit-serial hardware to do mathematical computations. Also, as with other projects [5, 12], RRANN uses look-up tables to implement the activation function (Equation (2)) and its derivative.

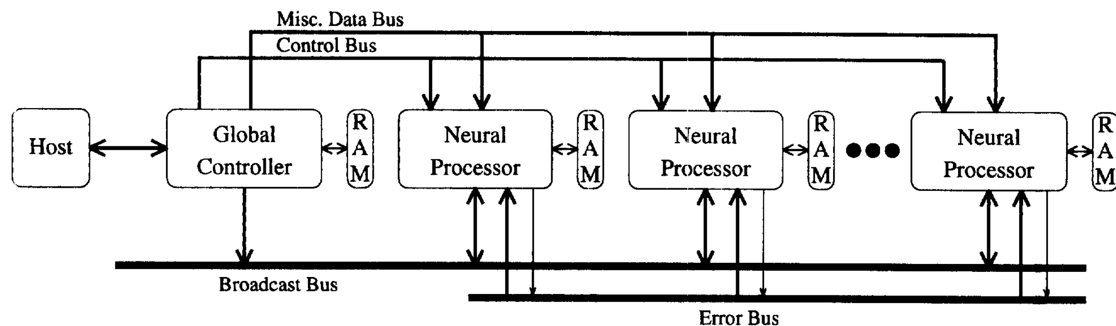

Figure 1: The General RRANN Architecture.

3.1 The Feed Forward Stage

To begin the feed forward stage, the global controller broadcasts an input value to the neural processors. This broadcast is done over a time-multiplexed bus one input value at a time [12]. As each input value is placed on the bus, the neural processors read it and do the appropriate weight multiplication for neurons in the first hidden layer. After the multiplication is complete, the product is accumulated with those from previous input iterations (Equation (1)), and the next input is broadcast over the bus. This method of interconnection allows for a large amount of parallelism to occur as all hardware neurons perform a multiply concurrently. It also allows for a large amount of scalability because hardware neurons can be added to the network by simply connecting them to the bus. Also, the hardware neuron only requires one multiplier; this allows for the development of one generic neuron that can be used with networks of arbitrary size.

After the evaluation of the first hidden layer is completed, the neural processors become responsible for broadcasting activation values to be used as inputs to the next layer. Before the evaluation of another layer begins, the hardware neurons buffer the *net* values calculated for the first hidden layer. Then, each hardware neuron in turn evaluates (2) for its buffered *net* value and broadcasts the result over the broadcast bus to all other hardware neurons including itself. This process iterates as the hardware neurons evaluate (1) and buffer the new *net* values as was done for the first hidden layer.

If the network has four layers, the above process is again repeated for the output layer. After the *net* values for the output layer have been found, they are used to evaluate (2), and the results are compared with the input pattern's target outputs (T_i). These compare operations ($T_i - O_i$) are computed in parallel, and the differences are buffered for use by the backpropagation stage.

3.2 The Backpropagation Stage

The backpropagation stage begins by finding the errors for the output layer using the differences found above according to (3). This operation can be executed with a large degree of parallelism, is purely local, and, therefore, does not require any communication of data between hardware neurons.

Erdogan and Hong [13] point out a stumbling block to parallelism in the calculation of (4). This problem occurs when error values are broadcast back through the network in a similar manner as activation values during the feed forward stage. The weight values needed for the multiplication after the broadcast are not locally available to the hardware neuron that needs it. Erdogan and Hong suggest as a possible solution the duplication of weight values. RRANN solves the problem in a fundamentally different way that avoids weight duplication and error broadcasting.

RRANN completes the computation of Equation (4) for each hidden neuron in the network (including the summation) before beginning (4) for another hidden neuron. This operation is illustrated by Figure 2. It begins as each output neuron calculates the error-weight product ($\delta \cdot W$) in parallel. Each neural processor then sums the products generated by its hardware neurons, and this partial sum is then communicated to the targeted neural processor. This target neural processor receives these partial sums and then completes (4) by adding the partial sums together and multiplying the appropriate activation derivative.

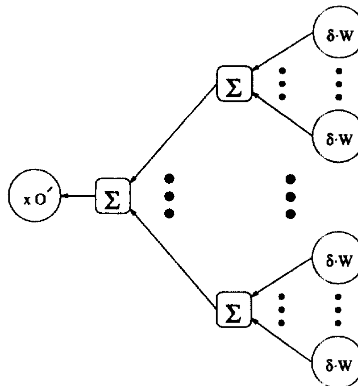

Figure 2: RRANN's Method of Computing Errors for Hidden Neurons.

Computing Equation (4) in this manner achieves the same level of parallel execution as the feed forward and update stages. This is an important advantage that overcomes the problem of non-local weight distribution discussed by Erdogan and Hong [13] without requiring weight duplication or broadcasting. However, a small amount of scalability is lost as each neural processor requires its own line of communication. As the number of neural processors grow, the number of error communication lines grow. This increase in communication lines also necessitates an increase in adder circuitry for merging the partial sums. These requirements are minimized, however, by the use of bit-serial data representation. The communication lines are only one bit wide, and bit-serial adders only require one Xilinx Combinational Logic Block (CLB).

3.3 The Update Stage

The update stage begins in a similar manner to the feed forward stage with the global controller broadcasting input values to the first hidden layer. The hardware neurons then use the activation and error values found by the first two stages to compute (5). The ΔW produced is immediately used in the calculation of (6), and the new weight value is copied over the old value. This process is repeated for every weight in the hidden layer in a parallel fashion as each input presents its value. Then RRANN begins changing the weights between the first hidden layer and the output (in a three layer network) or the second hidden layer (in a four layer network). This is again repeated, in the case of a four layer network, for the weights between the second hidden layer and the output layer.

The completion of the update stage marks the end of the evaluation of one complete training pattern. The three stages are repeated for each pattern in the training set (*one epoch*), and new epochs are run until the network is sufficiently trained.

4 Implementation and Performance

The RRANN architecture has been built and tested using a commercially available FPGA board (The SCA board) produced by National Technology Inc. All circuitry was designed and simulated using Mentor Graphic's schematic capture software, and these circuits were implemented on FPGAs using Xilinx's XACT software. By the use of a novel technique known as run-time reconfiguration, six hardware neurons were implemented on each Xilinx XC3090 FPGA [11]. The RRANN circuitry successfully simulated at 10 MHz and operated at 14 MHz. With improved pipelining, especially between the neural processor and local RAM, it is estimated that the circuitry could operated in excess of 40 MHz.

Run-time reconfiguration gives the user three separate options for implementing the RRANN architecture. The first option configures all three stages of operation on the FPGAs at the same time. The second option configures the feed forward and update stages on the FPGAs and then *reconfigures* the FPGAs when the

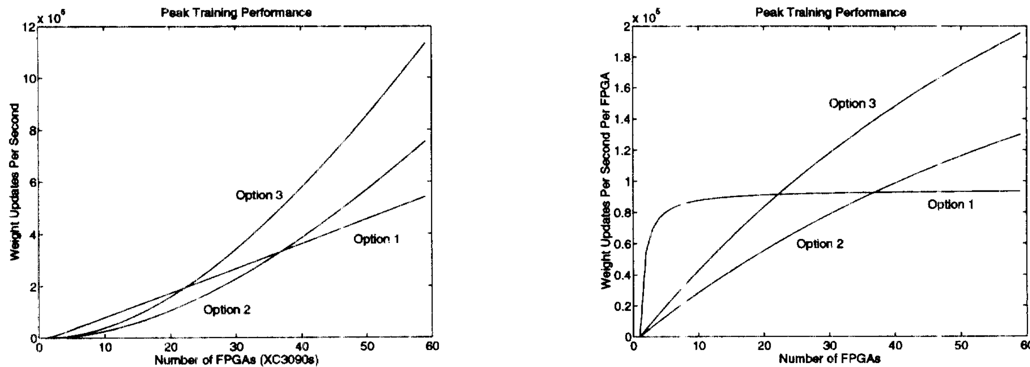

Figure 3: RRANN's Training Mode Performance (14 MHz).

backpropagation stage is to be executed. The third option configures only one of the three stages on the FPGAs at a time and reconfigures the FPGAs with the next stage in the sequence whenever one stage completes. The first two options have been studied theoretically, and the third option has been designed, implemented, and tested.

The number of hardware neurons per neural processor (FPGA) increases with each successive option. This is a result of better FPGA utilization because the hardware for inactive stages is occupying fewer FPGA resources. It is estimated that only one hardware neuron per neural processor is possible with option one, and four hardware neurons per neural processor with option two. Option three, however, made possible the implementation of six hardware neurons per neural processor.

This improved utilization, however, is not free. The time it takes to reconfigure the FPGAs uses time that could otherwise be spent executing the backpropagation algorithm. RRANN's performance for backpropagation learning (measured in weight updates per second) is summarized in Figure 3 as a function of the number of FPGAs (neural processors) available. The figure shows that because of reconfiguration overhead, option one performs better when the number of FPGAs is small (< 23). As the number of FPGAs increases beyond 23, option three begins to provide increasingly better performance. Figure 4 shows the feed forward mode performance for operational mode (feed forward only). This is measured in connections per second, and the figure shows that option three is clearly better. This result is not surprising because only the feed forward stage is needed, and reconfiguration is eliminated.

5 Conclusions

The RRANN architecture is an original neural network architecture developed by the authors to speed-up neural network execution by utilizing the natural parallelism of the backpropagation algorithm. Attempts were made to minimize hardware requirements by using bit-serial arithmetic and look-up tables for evaluation of the activation function. The architecture contained three stages of operation, feed forward, backpropagation, and update, and there were three options for implementing the stages depending on the number of FPGA reconfigurations done.

Interconnections between adjacent layers in the network are done over one time-multiplexed bus (broadcast bus). Each neuron in a layer communicates its activation in turn to the next layer over this bus. The advantages of this approach include the fact that only one generic neuron with one multiplier needs to be developed. This generic neuron can be used for networks of arbitrary size allowing RRANN to be very scalable. Also, parallelism is exploited in all three stages of the backpropagation algorithm.

During the backpropagation stage, the multiplication of errors and weights are done before communication to maintain a high level of parallelism. This operation overcomes the problem of distributing non-local weights while maintaining the high level of parallelism achieved during feed forward and update, and it does

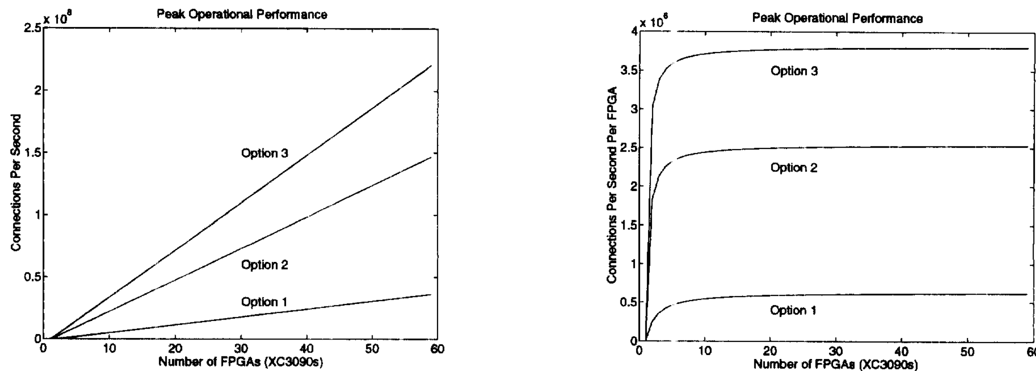

Figure 4: RRANN's Operational Mode Performance (14 MHz).

so without the duplication or communication of weights. This solution does compromise some amount of scalability, but this compromise is minimized because of bit-serial data paths and adders, and the computation of partial sums before communication. Three different options for implementing RRANN exist, each utilizing different levels of run-time reconfiguration. The third option was designed and implemented, and the performance for all three were presented and compared.

References

[1] E. Collins, S. Ghosh, C. Scofield, An Application of a Multiple Neural Network Learning System to Emulation of Mortgage Underwriting Judgements, *IEEE International Conference on Neural Networks*, 2:459-466, 1988.

[2] P. F. Castelaz, Neural Networks in Defense Applications, *IEEE International Conference on Neural Networks*, 2:473-480, 1988.

[3] T. Sejnowski and C. Rosenberg, NETtalk: A Parallel Network that Learns to Read Aloud, *Johns Hopkins University, Electrical Engineering and Computer Science Technical Report JHU/EECS-86/01*, 1986.

[4] D. G. Bounds, P. J. Lloyd, and G. Waddell, A Multilayer Perceptron Neural Network for the Diagnosis of Low Back Pain, *IEEE International Conference on Neural Networks*, 2:481-490, 1988.

[5] T. Watanabe, et. al., Neural Network Simulation on a Massively Parallel Cellular Array Processor: AAP-2, *IEEE International Joint Conference on Neural Networks*, 2:155-61, Washington DC, 1989.

[6] T. Nordstrom, and B. Svensson, Using and Designing Massively Parallel Computers for Artificial Neural Networks, *Journal of Parallel and Distributed Computing*, 14:260-285, 1992.

[7] W. A. Fisher, R. J. Fujimoto, and R. C. Smithson, A Programmable Analog Neural Network Processor, *IEEE Transactions on Neural Networks*, 2:222-229, 1991.

[8] S. Satyanarayana, Y. P. Tsividis, and H. P. Graf, A Reconfigurable VLSI Neural Network, *IEEE Journal of Solid-State Circuits*, 27:67-81, 1992.

[9] A. F. Murray, Pulse Arithmetic in VLSI Neural Networks, *IEEE Micro* 64-74, Dec. 1989.

[10] J. Tomberg and K. Kaski, Digital VLSI Architecture of Backpropagation Algorithm with On-Chip Learning, *Artificial Neural Networks. Proceedings of the 1991 Internation Conference. ICANN-91*, 2:1561-4, Espoo, Finland, 1991.

[11] J. Eldredge and B. Hutchings, Circuit Density Enhancement Through FPGA Run-Time Reconfiguration, Submitted to *IEEE Workshop on FPGAs for Custom Computing Machines*, Napa, CA, 1994.

[12] D. Hammerstrom, A VLSI Architecture for High-Performance, Low-Cost, On-chip Learning, *IEEE International Joint Conference on Neural Networks*, 2:537-544, San Diego CA, 1990.

[13] S. S. Erdogan and T. H. Hong, Massively Parallel Computation of Back-Propagation Algorithm Using The Reconfigurable Machine, *World Congress on Neural Networks '93*, 4:861-4, Portland OR, 1993.